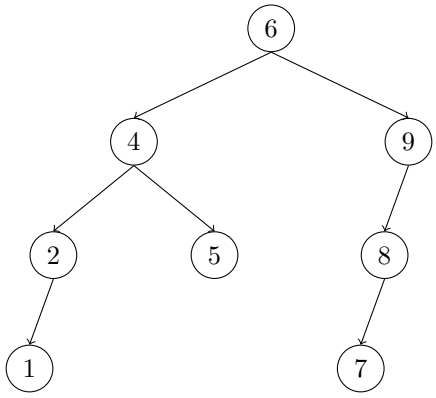
CS 61BL Summer 2021

Trees and Tree Traversals

Recurring Section 7: Tuesday July 20, 2021

1 Tree-versal



a) What is the pre-order traversal of the tree?

6 4 2 1 5 9 8 7

b) What is the post-order traversal of the tree?

1 2 5 4 7 8 9 6

c) What is the in-order traversal of the tree?

1 2 4 5 6 7 8 9

d) What is the level-order traversal of the tree?

6 4 9 2 5 8 1 7

2 Runtime Questions

Provide the best case and worst case runtimes in theta notation in terms of N, and a brief justification for the following operations on a binary search tree. Assume N to be the number of nodes in the tree. Additionally, each node correctly maintains the size of the subtree rooted at it. [Taken from Final Summer 2016]

```
boolean contains(T o); // Returns true if the object is in the tree
```

Best: $\Theta(\quad)$ Justification:

Worst: $\Theta(\quad)$ Justification:

Solution:
Best: $\Theta(1)$ Justification: If the object is at the root.
Worst: $\Theta(N)$ Justification: If the object is at the leaf of a spindly tree.

```
void insert(T o); // Inserts the given object.
```

Best: $\Theta(\quad)$ Justification:

Worst: $\Theta(\quad)$ Justification:

Solution:
Best: $\Theta(1)$ Justification: One example may be inserting to the left child of the root of a right leaning spindly tree.
Worst: $\Theta(N)$ Justification: One example may be inserting to the leaf node of a right leaning spindly tree.

```
T getElement(int i); // Returns the ith smallest object in the tree.
```

Best: $\Theta(\quad)$ Justification:

Worst: $\Theta(\quad)$ Justification:

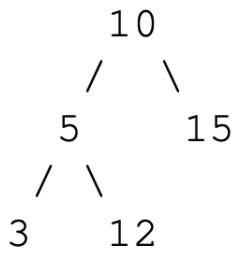
Solution:
Best: $\Theta(1)$ Justification: One example may be if `i = 1` and the tree is a very spindly right leaning tree.
Worst: $\Theta(N)$ Justification: One example may be if `i = N` and the tree is a very spindly right leaning tree.

3 Is This a BST?

The following code should check if a given binary tree is a BST. However, for some trees, it returns the wrong answer. Give an example of a binary tree for which `brokenIsBST` fails.

```
public static boolean brokenIsBST(TreeNode T) {
    if (T == null) {
        return true;
    } else if (T.left != null && T.left.val > T.val) {
        return false;
    } else if (T.right != null && T.right.val < T.val) {
        return false;
    } else {
        return brokenIsBST(T.left) && brokenIsBST(T.right);
    }
}
```

Solution: Here is an example of a binary tree for which `brokenIsBST` fails:



Explanation: The method fails for some binary trees that are not BSTs because it only checks that the value at a node is greater than its left child and less than its right child, not that its value is greater than every node in the left subtree and less than every node in the right subtree. Above is an example of a tree for which it fails. It is important to note that the method does indeed return true for every binary tree that actually is a BST (it correctly identifies proper BSTs).

Now, write `isBST` that fixes the error encountered in part (a).
Hint: You will find `Integer.MIN_VALUE` and `Integer.MAX_VALUE` helpful.

```
public static boolean isBST(TreeNode T) {
    return isBSTHelper(                                          );
}

public static boolean isBSTHelper(                                ) {
```

}

Solution:

```
public static boolean isBST(TreeNode T) {
    return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

public static boolean isBSTHelper(TreeNode T, int min, int max) {
    if (T == null) {
        return true;
    } else if (T.val < min || T.val > max) {
        return false;
    } else {
        return isBSTHelper(T.left, min, T.val)
                && isBSTHelper(T.right, T.val, max);
    }
}
```

4 Pruning Trees

Assume we have some binary search tree, and we want to prune it so that all values in the tree are between L and R, inclusive. Pruning simply means removing certain items and adjusting the tree so that it is still a BST. Fill out the method below that takes in a BST, as well as L and R, and returns the pruned tree. Note that the root of the original tree might not be between L and R, so make sure you return the root of the new pruned tree.

```
class BST {
    int label;
    BST left; // null if no left child
    BST right; // null if no right child
}

public BST pruneBST(BST root, int L, int R) {
    if (________________) {
        return ________;
    } else if (____________________) {
        return pruneBST(____________, _____, _____);
    } else if (____________________) {
        return pruneBST(____________, _____, _____);
    }
    ____________ = pruneBST(____________, _____, _____);
    ____________ = pruneBST(____________, _____, _____);
    return _______;
}
```

Solution:

```
public BST pruneBST(BST root, int L, int R) {
    if (root == null) {
        return null;
    } else if (root.label < L) {
        return pruneBST(root.right, L, R);
    } else if (root.label > R) {
        return pruneBST(root.left, L, R);
    }
    root.left = pruneBST(root.left, L, R);
    root.right = pruneBST(root.right, L, R);
    return root;
}
```

5 BTree Motivation

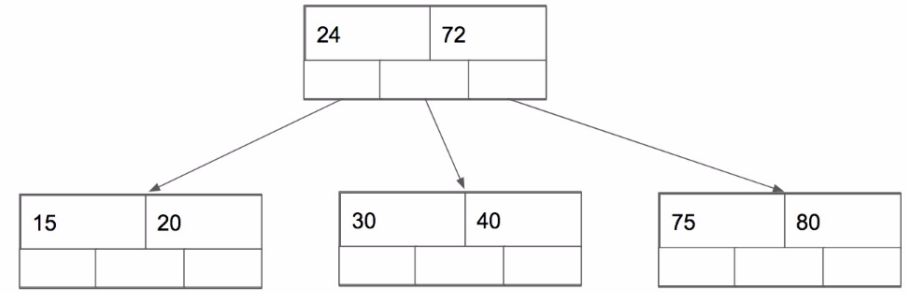
1. Why does a binary search tree have a worst case runtime of $\theta(n)$ for `contains`?

   If the search tree is not bushy, i.e. is just a line of nodes, we get very poor performance. Another way of thinking about this is if a BST becomes a line of nodes, it will simply be a `Linked List` and have the runtime of such.

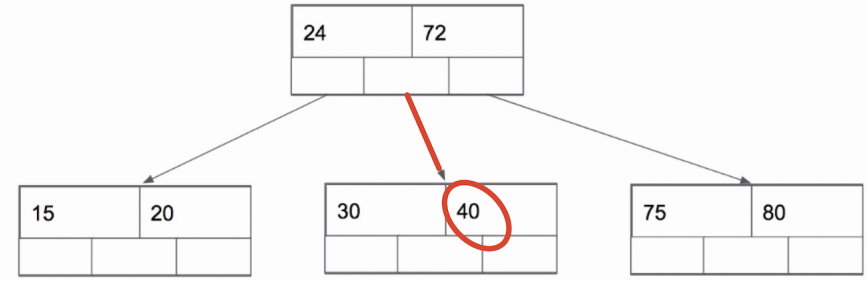
2. Give a sequence of operations, such that if they were inserted in the order they appear, would result in a "poor" binary search tree.

   Any increasing or decreasing sequence will work, e.g. `1, 2, 3, 4, 5`.

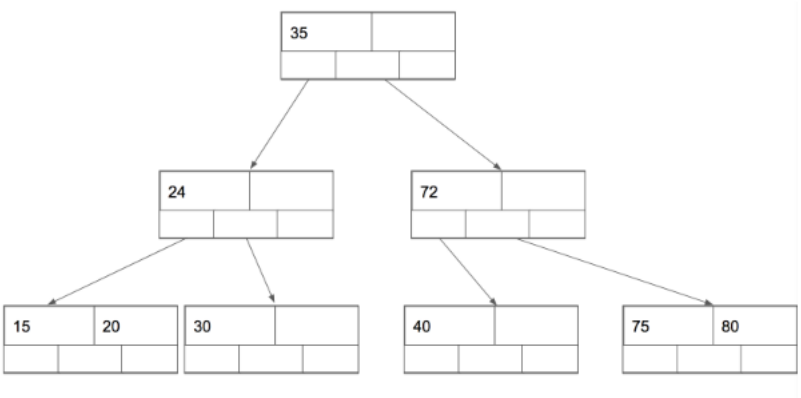
3. Examine this B-tree with order 3. Mark the paths taken when the user calls `contains(40)`.



   We examine the root node and see that 40 is greater than 24 and less than 72, so we take the middle edge to the child node. We examine this node and find 40.



4. Now call `insert(35)`, and draw the resulting tree.



5. What property of a B-tree rectifies problems of binary search trees, such as the one in 1.1? Why would you not use a B-tree?

   B-trees are balanced - because we always split nodes on insertion and move keys upwards in the tree, we ensure we never get the "long tail" of nodes that can occur in a normal binary search tree. That ensures we get $O(logn)$ performance. Another benefit is that because we store more elements at a node, we have to do fewer traversals in the tree. B-trees are significantly more complicated than binary search trees and more difficult to implement. Red black trees provide a way for us to implement B-Trees in a simpler way, without losing the advantages of the B-Tree.